DISTRIBUTED SEQUENTIAL COMPUTING

LEI PAN, LUBOMIR F. BIC, MICHAEL B. DILLENCOURT, AND MING KIN LAI *

Abstract. Sequential computations can benefit from distributed computing through the use of self-migrating threads. On large problems, significant performance improvement is achieved by eliminating disk paging completely, trading off against a modest amount of data communication. The key idea is to have computation and small-sized data move to meet with large-sized data rather than the other way around. Using self-migrating threads with strong mobility provides a flexible way of grouping computations into programs. One of the benefits of this flexibility is ease of application-level programming. Sequential programs are easily augmented into distributed sequential programs for solving large problems, and the distributed sequential programs serve as a good starting point for incremental parallelization. Using our self-migrating threads to implement distributed sequential programs is almost as efficient as using message passing, because we have made thread migration almost as efficient as message passing.

Key words. Distributed sequential computing, Navigational programming, Self-migrating thread, Computation mobility, Principle of pivot-computes, Algorithmic integrity, Incremental parallelization

1. Introduction.

One of the major goals of distributed computing is using networks of workstations to solve large problems quickly. The most obvious way of achieving this is through parallel computation, when the problem lends itself to parallelization. This can make the computation faster in several ways. Speedup is achieved by distributing the total computational work over multiple workstations; this is the fundamental advantage of parallelism. But in addition, the computation on each workstation may use a smaller working set, leading to fewer or no page faults. Since dealing with page faults is part of the work that must be performed, distributing a computation over multiple workstations thus decreases the total amount of work required by the computation. In some cases, the efficiency advantages due to the second effect—reducing total work—may dwarf the efficiency advantages due to distributing the workload. This observation suggests that even for algorithms that are not easily parallelizable, distributing the data and the sequential computation over multiple workstations can potentially result in significant increases in efficiency. In this chapter, we consider **distributed sequential computing** (**DSC**), defined as computing with distributed data using a single locus of computation, and describe a new approach to DSC based on the use of self-migrating threads.

In order to be practically useful, DSC must be efficient and scalable, and it must also be easy to program. Neither of the two classical approaches to distributed programming—Distributed Shared Memory (DSM) or Message Passing (MP)—satisfies both requirements. DSM is easy to use, but it is inefficient and unscalable [23]: the gain from eliminating disk paging may be overshadowed by the cost of network communication. MP is efficient and scalable, but it is in general hard to use [16]. MP is particularly ill-suited for DSC, for reasons that will be explained later on. Our approach to DSC is based on the use of self-migrating threads. The basic idea is to have the sequential computation, carried by a self-migrating thread, run on a network of workstations. The data is distributed; in essence, we use the network as a data farm. Rather than having the data migrate to the computation, the computation (not code) follows the data. The programming of self-migrating threads is called **Navigational Programming** (**NavP**).

DSC when implemented using the NavP approach has two major advantages:

1. Improved and Scalable Performance: The overhead of disk paging is eliminated completely: we can achieve this simply by making sure that each data partition fits in the main memory of a workstation. Communication through the network is handled by the

*School of Information and Computer Science, University of California, Irvine, Irvine, CA 92697-3425, USA ({pan,bic,dillenco,mingl}@ics.uci.edu) self-migrating threads that carry small-sized data to meet with large-sized data in every sub-computation of the entire execution. Moreover, the thread doing the computation can send out auxiliary threads to other machines to post-write computed data and pre-fetch data for subsequent computation. In this way, the DSC program works even when the amount of data that needs to be processed is much larger than the amount of memory available in the entire network.

2. Good Programmability: An existing sequential algorithm does not have to be substantially rewritten; it merely needs to be augmented with navigational statements (e.g., *hop*()) to migrate the computation from one machine to another at the appropriate points in the algorithm. As described above, pre-fetching and post-writing may be done in parallel with the computation, but this parallelism is very easy to introduce and does not require modifying the sequential algorithm being executed. The thread doing pre-fetching and post-writing typically consist of only a few lines of code, and their synchronization with the main computing thread is straightforward.

We describe DSC, and the reasons why it is not a good fit with DSM or MP, in §2. Our NavP approach to DSC, along with some simple examples, is presented in §3. Section 4 contains three case studies of DSC: matrix multiplication and two algorithms for solving linear system of equations. A comparison of other possible approaches with ours is given in §5. Section 6 provides some additional remarks.

2. Distributed Sequential Computing.

We define **distributed sequential computing (DSC)** as computing with distributed data using a single locus of computation. There are several reasons for using sequential computing in a distributed environment. First, only very few algorithms (referred to as "embarrassingly parallel") can be perfectly parallelized. Many parallel algorithms have inherently sequential portions, often over distributed data, that become bottlenecks for speedup (this phenomenon is the basis of Amdahl's law and its variants), so it is important to be able to handle these sequential portions efficiently and easily. Second, when solving problems on distributed-memory systems, programmers may choose to decompose a problem into coarse-grained sequential sub-tasks that run in parallel, even if fine-grained data parallelism is an option [16]. Third, distribution and parallelization are two major components in distributed parallel programming [14]. The job of distribution is to map the task onto different machines so that the overall communication overhead is minimized, while the job of parallelization is to identify sub-tasks that can be performed concurrently. If we can find an easy and efficient approach to handle distribution for distributed sequential computing, we can then build parallelism using concurrent DSC sub-tasks in an incremental fashion. Finally, DSC programs can be used to solve large problems when parallelization is impossible or unaffordable for any reasons [1]. It is important to note that DSC is not being proposed as a competitor to parallel computing; rather, it is a complementary programming method that can be used in conjunction with parallelism.

Both DSM and MP fail to provide good underlying support for efficient, scalable, and easily developed DSC programs. A single process running on DSM is, conceptually at least, a DSC program, but it is not scalable. This is because although disk paging is eliminated, new communication through the network is introduced, and the size of the communicated data is proportional to the network memory farm and can be arbitrarily large compared to the main memory on the workstation on which the process is running. The DSM approach moves more data than needed. Moreover, when the size of the problem working set is larger than that of the main memory, thrashing is going to happen no matter what paging device is being used; paging to remote main memory does not reduce the number of page faults, it only improves the service time for each page fault. In fact, a special case of DSM, "remote main memory paging," works under the assumption that the working set of a problem cannot be larger than the main memory of the workstation [4].

Message passing provides efficient and scalable programs, but distributed programming with MP is hard, and is not amenable to incremental parallelization. The transformation from a sequential program into an MP program is an "abrupt break" since the code must be completely restructured [16]. So in situations where full parallelization is impossible or too expensive, re-implementation of a sequential algorithm in MP is usually inappropriate, as it requires paying the cost of re-implementing the program without gaining the advantage of parallelism.

To be viable, distributed sequential computing must be both scalable and easy to use. The key to scalability in distributed computing is reducing the amount of data being moved. The principle of **pivot-computes**, defined as the principle under which a (sub-)computation takes place on the node that owns the large-sized data, is necessary to achieve the goal of minimizing communication. The node that owns the large-sized data is called the **pivot node**. The scope of the principle of pivot-computes is code building blocks that can have one or more statements. It is obvious that DSM-based DSC violates the principle of pivot-computes because at any point of execution the large-sized data can be anywhere in the workstation farm but the stationary sequential process always pulls in the data, large or small in size, from the farm for computation. This explains why DSM-based DSC is not scalable.

Moving computation locus across machine boundaries is unavoidable if we follow the principle of pivot-computes in distributed memory. This observation is an immediate consequence of the following two basic facts: (1) in order for a computation to be performed on some data pieces, the data pieces and the locus of computation need to be together; and (2) the pivot nodes that host large-sized distributed data pieces change as different sub-computations of a program are carried out. We define **computation mobility** as the ability for the locus of computation to migrate across distributed memories and continue the computation as it meets the required data. This migration is controlled by a programmer either explicitly using navigational statements in the programs or implicitly through data distribution.

Computation mobility can be supported in at least two ways. Message passing can be used to transfer locus of computation from one machine to another. In fact, since the metaphor of message passing suggests pivot-computes strongly, an MP programmer usually does not make the mistake of moving large-sized data. This is why MP is efficient and scalable, and hence popular in the high-performance commercial software market [10]. But, moving locus of computation from one machine to another with MP is cumbersome. In an SPMD (single program multiple data) style, an MP program imposes "artificial constructs" that are usually in the form of **If/Else** blocks that define which line of code runs on which workstation according to how data is distributed in the network. Thus, with MP, distributing data means restructuring code. When transforming a sequential algorithm to an MP implementation, code restructuring can be in the form of reordered and regrouped code lines that are assigned to different artificial constructs, or broken small loops that each runs on a different machine but altogether mimic the original large loop spanning the entire data. This code restructuring usually causes the MP implementation to look dramatically different from the original algorithm, and it is unnecessary because it does not contribute directly to performance improvement. Moreover, explicit synchronization is needed among the processes even if the MP program is a (distributed) sequential application.

Self-migrating threads that provide strong mobility [5] are another means to facilitating computation mobility. A NavP programmer can "drive" the computation to wherever wanted by augmenting the code with navigational statements (e.g., $hop()$). These augments will keep the original code structure of the sequential program unchanged, thus making code development and maintenance easy. Small-sized data can be "carried" to meet with large-sized data in the workstation farm. This makes it possible to follow the principle of pivot-computes and hence achieve scalability.

3. The Navigational Programming Approach. Self-migrating threads are conceptually similar to mobile software agents [15, 17, 13]. Mobile agents are programs that move autonomously among networked machines, carrying their data and execution states. A mobile agent, with strong mobility [5], can halt its execution, encapsulate the values of its variables, move to another machine, restore the state, and continue executing. This sequence of actions is referred to as a *hop*() statement.

Navigational programming using self-migrating threads with strong mobility provides a flexible and powerful way of grouping distributed computations into programs. Computations, represented by code lines in a program, can be "chained" using *hop*() statements. This allows programs to execute a particular computation on one participating node, then a second computation on another node, and so forth. In contrast, the SPMD programming style used by the MP approach has only one way to group computations. That is, computations are grouped by their execution locations, and the different groups are then put together using **If/Else If** constructs. This flexibility of the NavP approach simplifies the programming task considerably. Transforming a sequential algorithm into its NavP-based DSC implementation requires introducing only minor changes, namely the insertion of the appropriate navigational statements. The original code structure is preserved. We say that such a transformation preserves **algorithmic integrity**.

Of course, algorithmic integrity is beneficial only if the original code structure has the right properties (e.g., locality of reference). For this reason, it may be useful to rearrange the sequential code before inserting the navigational commands. The important point is that once a good starting point is chosen, there is no reason why the conversion from sequential code to DSC code requires making unnecessary or undesirable changes in the code structure.

3.1. A brief overview of the MESSENGERS system. All of the examples of DSC presented in this chapter were implemented using the MESSENGERS mobile-agent system [6, 7]. In this system, MESSENGERS code (with strong mobility) is translated by the MESSENGERS compiler into C code communicating using message passing with sockets. As part of this translation, the MESSENGERS program is broken into small C functions, so that each small function represents a unit of computation [25]. In essence, the navigational statements form the boundaries of these small functions. This C code is then further compiled into machine native code for execution. The MESSENGERS system allows code to be either loaded from a shared disk or, in a non-shared file system, sent across the network at most once, irrespective of how many times the locus of computation moves across the network [8].

As a result of this architecture, the additional overhead incurred by thread migration is quite small. When a thread migrates, the state of its computation needs to be sent over the network but the code itself does not. In the MESSENGERS implementation, thread migration is almost as efficient as message passing. The additional overhead of a *hop*() (other than the data, which would also have to be sent in a message-passing computation) is about 200 bytes [25].

3.2. Two simple examples. We use two simple and somewhat artificial examples to illustrate our NavP-based DSC approach. More realistic examples are provided in §4. In our first example, we assume that A and B are two large $N \times N$ matrices and we compute the vector $A\,(B\,diag(A))$, where $diag(A)$ is a vector consisting of the diagonal entries of A and both multiplications represent a matrix multiplied by a vector. We assume there are two nodes, with A located on node0 and B located on node1. The pseudocode is listed in Fig. 3.1(a). The diagonal entries of matrix A are extracted and assigned to a vector $v1$ of size N at line (1). Matrix B is then multiplied by $v1$ to obtain a vector of size N assigned to $v2$ at line (2). Finally, A is multiplied by $v2$ to obtain $v3$ at line (3). In the MP version, listed in Fig. 3.1(b), lines (1) and (3) are executed on node0 because they both require

```
(1)    $v1 = diag\,(A)$
(2)    $v2 = B\,v1$
(3)    $v3 = A\,v2$
```

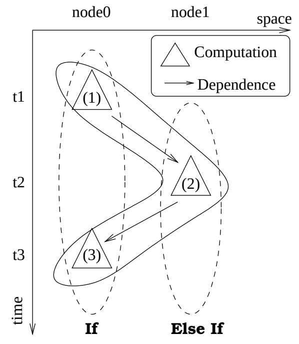
(a)

```
(1)    $v1 = diag\,(A)$
(1.1)  $hop\,(node1)$
(2)    $v2 = B\,v1$
(2.1)  $hop\,(node0)$
(3)    $v3 = A\,v2$
```

(c)

```
(0.1)If $\mu == node0$
(1)      $v1 = diag\,(A)$
(1.1)    $Send\,(v1,\ node1)$
(1.2)    $Recv\,(v2,\ node1)$
(3)      $v3 = A\,v2$
(3.1)ElseIf $\mu == node1$
(3.2)    $Recv\,(v1,\ node0)$
(2)      $v2 = B\,v1$
(2.1)    $Send\,(v2,\ node0)$
(2.2)EndIf
```

(b)



(d)

Fig. 3.1. *Computing on distributed data. (a) Sequential. (b) MP. (c) NavP. (d) Two ways of grouping computations.*

matrix A, while line (2) is on node1 because it computes with B. As a result, the three lines in the original algorithm is now reordered and regrouped - lines (1) and (3) are in one group and line (2) is in another, as depicted by the ovals in dashed lines in Fig. 3.1(d). The groups are then assigned to one of the **If**/**Else If** constructs (μ is the ID of the current node). This demonstrates that with MP distributing the data requires restructuring the code. Communication and synchronization between the two nodes are done through *Send* () and *Recv* () (lines (1.1), (1.2), (3.2), and (2.1)). Explicit synchronization is needed among the computations even if the MP program is (distributed) sequential. This MP implementation follows the principle of pivot-computes. The pivot nodes for lines (1) and (3) are node0, and for line (2) is node1. In the NavP implementation, with code shown in Fig. 3.1(c), $v1$ and $v2$ are **agent variables**, which are "carried" by a mobile agent to whichever node it visits, and A and B are **node variables**, which are stationary to nodes, on node0 and node1 respectively. The NavP code is basically the same as the sequential code except for the insertion of two *hop*() statements, which enable the thread to navigate between the two nodes. The NavP implementation also follows the principle of pivot-computes, and the amount of communication incurred, namely vectors $v1$ and $v2$, is the same as in the MP code. The NavP approach provides a flexible way of grouping computations into threads, as depicted in Fig. 3.1(d) by the angular shape in solid line. As a result, communication and synchronization among the sequential computations are intra-thread, and therefore are subsumed in the execution flow.

```
(1)   $For\ \ i = 2\,..\,n$
(2)       $A[i] = A[i-1] + 1$
(3)   $End\,For$
```

(a)

```
(1)    $For\ \ i = 2\,..\,n$
(1.1)    $x = A[i-1]$
(1.2)    If $i == n/2+1$  $hop\,(node1)$
(2)      $A[i] = x + 1$
(3)    $End\,For$
```

(c)

```
(0.1)If $\mu == node0$
(1)      $For\ \ i = 2\,..\,n/2$
(2)          $A[i] = A[i-1] + 1$
(3)      $End\,For$
(3.1)    $x = A[n/2]$
(3.2)    $Send\,(x,\ node1)$
(3.3)ElseIf $\mu == node1$
(3.4)    $Recv\,(x,\ node0)$
(3.5)    $A[1] = x + 1$
(1')     $For\ \ i = 2\,..\,n/2$
(2')         $A[i] = A[i-1] + 1$
(3')     $End\,For$
(3.6)EndIf
```

(b)

Fig. 3.2. *Looping over distributed array. (a) Sequential. (b) MP. (c) NavP.*

The second example is a simple sequential loop over a 1-D array $A[1..n]$, shown in Fig. 3.2(a). Because of the dependency between $A[i]$ and $A[i-1]$, this loop is hard to parallelize [18]. For data distribution, we assume the first $n/2$ elements of A are on node0 and the second $n/2$ elements on node1. The sequential MP code is shown in Fig. 3.2(b). The loop is broken into two parts in order to follow the principle of pivot-computes, one in **If** block (lines (1), (2), and (3)), another in **Else If** block (lines (1'), (2'), and (3')). At the array boundary with $A[n/2]$ on node0 and $A[n/2+1]$ on node1, communication of $A[n/2]$ and synchronization are needed (lines (3.2) and (3.4)). The NavP implementation, listed in Fig. 3.2(c), on the other hand, preserves the loop structure. The loop index i and the temporary variable x are agent variables. The arrays $A[.]$ on both nodes are two node variables acting as one distributed shared variable (DSV) [19].

4. Case Studies.

In this section, we present case studies with three real-world applications, namely matrix multiplication, Gauss-Seidel iteration, and Crout factorization. Our focus will be on the comparison of performance and programmability between the NavP-based and the other approaches to DSC.

When the total memory use of an application exceeds the size of the main memory on one node, virtual memory is used and hence disk paging occurs. When the working set cannot fit into the main memory, heavy disk paging, or thrashing, happens. Both paging and thrashing can dramatically degrade the performance, although the exact point at which the performance deteriorates may vary depending on the details of the implementation and the data access patterns. One way to reduce paging is to continue to use a sequential program and augment it with *ad hoc* "spill logic" that uses the knowledge of the future data access pattern to reduce disk paging overhead. But this solution is application dependent, cumbersome, and non-scalable.

To eliminate disk paging, we use network-connected machines, and code the applications using the NavP approach. The basic idea is simply to decompose the entire data into smaller pieces each fitting completely into the main memory of a workstation, and therefore all computations on all the machines will be paging free. Self-migrating threads carrying relatively small amounts of data, such as loop indices or intermediate result, hop among the machines to perform computations.

In addition to the performance gained from eliminating paging, the NavP approach is easy. In principle, it is possible to mimic the behavior of the NavP code with message passing, using stationary processes that transfer the locus of computation by sending wake-up messages that contain the state of the computation. In two of our examples (matrix multiplication and Crout factorization), we provide MP-based DSC pseudocode to illustrate the additional burden placed on the programmer when MP is used rather than self-migrating threads.

The NavP implementations used MESSENGERS. All the performance tests were run on SUN Sparc Ultra-1's with 64MB of main memory, 1GB of virtual memory, and 10Mbps of Ethernet connection. These workstations have a shared file system (NFS).

4.1. Matrix multiplication.

The first example is matrix multiplication, i.e., $C = AB$, where A, B, and C are assumed to be square dense matrices for simplicity. We use a block-fashion coarse-grained algorithm, and partition the matrices into p horizontal and vertical strips, as depicted in Fig. 4.2 with $p = 3$. Each piece C_{ij} is the product of two slices A_i and B_j. Sequential pseudocode is listed in Fig. 4.1(a).

The DSC implementation of matrix multiplication assumes that a sequence of sub-matrix multiplications are performed, and each multiplication in the sequence requires a pre-fetching (to obtain the two sub-matrices being multiplied) and a post-fetching (to store the resulting sub-matrix). Performing fetching in parallel with computations provides some speedup, even if the computations are strictly sequential. To explore further parallelism in

```
(1)   For $i = 0..p-1$
(2)       For $j = 0..p-1$
(3)           $C_{ij} = A_i\, B_j$
(4)       End For
(5)   End For
```

(a)

```
(1)    For $i = 0..p-1$
(2)        For $j = 0..p-1$
(2.1)          inject $(WR(i,j))$
(2.2)          waitEvent $(IO_{b(i,j)})$
(3)            $C_{ij} = A_i\, B_j$
(3.1)          hop $(!\mu)$
(4)        End For
(5)    End For
```

(b)

```
(1) WR (int $i$, int $j$)
(2)   hop $(!\mu)$
(3)   write $(C_{i''j''})$
(4)   read $(A_{i'}, B_{j'})$
(5)   signalEvent $(IO_{b(i',j')})$
(6) End
```

(c)

FIG. 4.1. *Pseudocode for matrix multiplication. (a) Sequential. (b) NavP. (c) Fetching.*

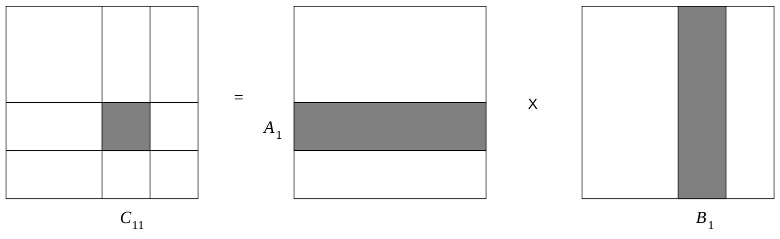


FIG. 4.2. *Matrix decomposition.*

matrix multiplication, one can use, e.g., Gentleman's algorithm. In the interest of clarity and simplicity in presenting the idea of DSC, we focus on the first source of parallelism coming from parallel fetching. We use two networked workstations. There are two threads carrying out the task. One thread called *Multiplier*, with the pseudocode listed in Fig. 4.1(b), calculates C_{ij} on the node where A_i and B_j reside. This thread hops between the two nodes alternately carrying its computation state, the loop indices i and j which indicate which C_{ij} is being calculated. The computation locus is only on a single node, either node0 or node1, at any particular time. A second thread, called *WR* (writer and reader, with code listed in Fig. 4.1(c)), injected (i.e., spawned) by the *Multiplier* at line (2.1) on one node, hops to the other node $(!\mu)$, post-writes $C_{i''j''}$ computed from the previous iteration to disk, and pre-reads $A_{i'}$ and $B_{j'}$, the next pair of slices to be computed after the pair A_i and B_j. Obtaining (i'', j'') or (i', j') from (i, j) is trivial, and the details are thus omitted. These two threads synchronize among themselves using local "events." For example, *Multiplier* will need to, on either node, wait for *WR* running on the same node to finish before it can start computing; so at line (2.2) it waits for an event, $IO_{b(i,j)}$, signaled by *WR* (at line (5)) after reading A_i and B_j. In MESSENGERS, *signalEvent*() and *waitEvent*() implement the classical operations of process blocking and wakeup. Only program state (e.g., loop indices i and j) that is tiny compared to the data (i.e., matrix slices), migrates between the two nodes. Provided the total size of A_i, B_j and C_{ij} does not exceed the size of the main memory on either node, the computation will not cause disk paging.

Fig. 4.3 lists the pseudocode for MP implementation. The code is much longer, and more importantly the code lines are regrouped according to where they are executed. For example, line (3) in the original code Fig. 4.1(a) is repeated in the MP implementation (lines (6) and (23) in Fig. 4.3) because this code line is executed on both nodes. The two nested loops are duplicated on both nodes, with the one on node1 being a *While*() loop that is a semantically equal translation of the loop nest. Here node0 has all the loop state information and node1 is acting as a slave by doing what it is told to do. Another way of implementing is to have both nodes compute redundantly the loop state information. In

```
(1)  If $\mu == node0$
(2)    $For\ i = 1..p$
(3)      $For\ j = 1..p$
(4)        If $(i * p + j)\%2 == 0$
(5)          $Send\,(\text{“}read\text{”}, (i, j), node1)$
(6)          $C_{ij} = A_i\, B_j$
(7)        Else
(8)          $Send\,(\text{“}comp\text{”}, (i, j), node1)$
(9)          $write\,(C_{i''j''})$
(10)         $read\,(A_{i'}, B_{j'})$
(11)       End If
(12)       $Recv\,(sync, node1)$
(13)     $End\ For$
(14)   $End\ For$
(15)   $Send\,(\text{“}stop\text{”}, (0, 0), node1)$
(16) ElseIf $\mu == node1$
(17)   $While\,(1)$
(18)     $Recv\,(s, (i, j), node0)$
(19)     If $s == \text{“}read\text{”}$
(20)       $write\,(C_{i''j''})$
(21)       $read\,(A_{i'}, B_{j'})$
(22)     Else If $s == \text{“}comp\text{”}$
(23)       $C_{ij} = A_i\, B_j$
(24)     Else If $s == \text{“}stop\text{”}$
(25)       $exit$
(26)     End If
(27)     $Send\,(sync, node0)$
(28)   $End\ While$
(29) End If
```

FIG. 4.3. *Pseudocode for matrix multiplication in MP.*

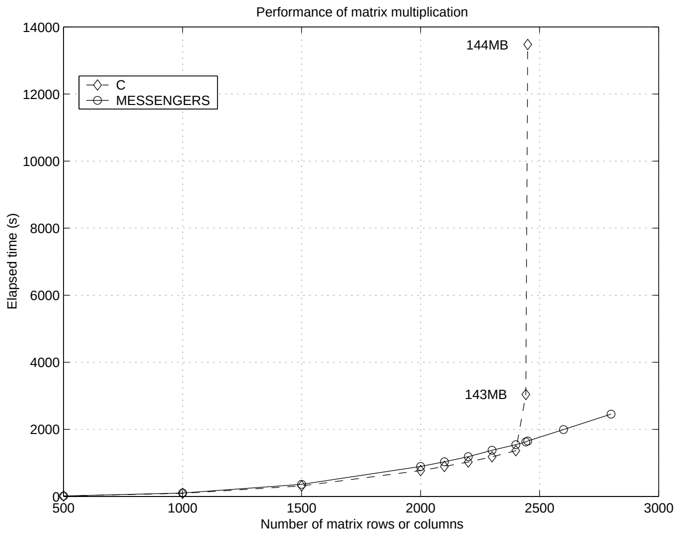


FIG. 4.4. *Performance of matrix multiplication.*

this way, the amount of communication will be slightly smaller, but both nodes will have to maintain two full loops. It is not clear which way is better, but in either case significant code restructuring is unavoidable.

While the MP pseudocode represents a significant departure from the original algorithm, the NavP implementation leaves the code structure essentially unchanged (Fig. 4.1(b)); the algorithm is augmented with three statements: a *hop*() (line (3.1)), a *waitEvent*() (line (2.2)), and an *inject*() (line (2.1)). The code for fetching (Fig. 4.1(c)) is written separately and it is not tangled with the code for computing at all.

Fig. 4.4 shows that the performance of the sequential C code deteriorates dramatically after the total size of the matrices reaches a certain critical value (around 140MB). Our NavP implementation has performance almost identical to that of the C implementation when the total size of the matrices is below this critical value, and it continues the same performance trend after the total size exceeds this critical value.

4.2. Gauss-Seidel iteration. Much of the CPU time spent in executing numerical analysis programs is used in solving linear systems of equations. In this and the next subsections we consider two classical methods for solving linear systems: namely, an iterative method based on Gauss-Seidel iteration and a direct method based on Crout factorization.

Let $Ku = f$ be a system of linear equations, where K is an $N \times N$ matrix, u and f are vectors of size N. Matrix K can be decomposed into $K = D - L - U$, where D is the diagonal of K, and $-L$ and $-U$ are the strictly lower and upper triangular parts of K. Gauss-Seidel iterative method [2] can be expressed as

$$u^{n+1} \leftarrow P_G u^n + (D-L)^{-1} f, \tag{4.1}$$

where $P_G = (D-L)^{-1}U$ is the Gauss-Seidel iterative matrix.

We update the components of the solution vector u in ascending order. Components of the new approximation u^{n+1} are used as soon as they are computed. In other words, we solve the j^{th} equation for u_j using new approximations for components $1, 2, ..., j-1$. The initial value of u can be arbitrary, e.g., $u^0 = [0, 0, ..., 0]^T$. Gauss-Seidel iteration can be done in a block fashion, in which only a portion of the solution vector u^{n+1} is updated given a slice of the matrix K and the entire solution vector from the previous iteration u^n, as illustrated in Fig. 4.5. This is a coarse-grained algorithm. The stopping criteria is $\|e^n\|_2 < TOL$, where $e^n = u^n - u^{n-1}$ is the error at iterative step n, TOL is a user defined tolerance, and $\|\cdot\|_2$ is the L^2 norm defined by $\|e\|_2 = \{\sum_{j=0}^{N-1} e_j^2\}^{\frac{1}{2}}$.

Since matrix K is usually banded, we do not store the leading and trailing zeros of the rows. This is referred to as the "compact row storage scheme," under which the rows are stored in a 1-D array, and an integer array of size N is used to store the positions, in the 1-D array, of the first nonzero terms of all rows.

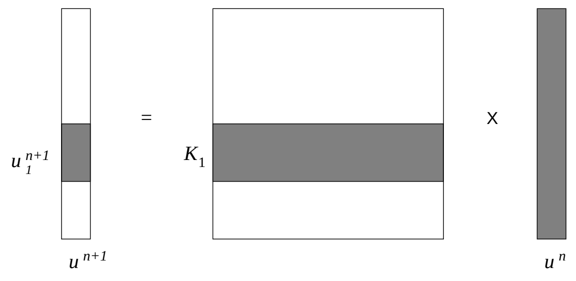


FIG. 4.5. *Matrix decomposition and block fashion of Gauss-Seidel iteration.*

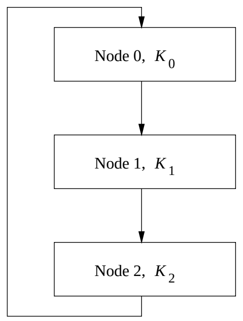


FIG. 4.6. *The network for Gauss-Seidel iteration.*

```
(1)   $u_1 = \{0\}$; $err = 1$
(2)   $While\ err > TOL$
(3)       $u_0 = u_1$
(4)       $For\ j = 0\,..\,p-1$
(5)           $u_1 = block\_gs\,(K_j,\, u_0,\, f)$

(6)       $End\ For$
(7)       $err = check\_error\,(u_0,\, u_1)$
(8)   $End\ While$
```

(a)

```
(1)   $u_1 = \{0\}$; $err = 1$
(2)   $While\ err > TOL$
(3)       $u_0 = u_1$
(4)       $For\ j = 0\,..\,p-1$
(5)           $u_1 = block\_gs\,(K_j,\, u_0,\, f)$
(5.1)         $hop\,((j+1)\%p)$
(6)       $End\ For$
(7)       $err = check\_error\,(u_0,\, u_1)$
(8)   $End\ While$
```

(b)

FIG. 4.7. *Pseudocode for Gauss-Seidel iteration. (a) Sequential. (b) NavP.*

The matrix K is partitioned into horizontal slices, as illustrated in Fig. 4.5. Each slice of K is used to update the components of u corresponding to row positions of the slice. The network used by the NavP implementation is a ring, where the number of nodes is equal to the number of slices, p, into which K is partitioned; the case of $p = 3$ nodes is shown in Fig. 4.6. A thread carrying the solution vector u can visit a node, use the slice of K on that node to update the corresponding entries of u, and then hop to the next node, repeating this until the error satisfies the stopping criteria.

The pseudocode is shown in Fig. 4.7. The function *block_gs*() runs one step of Gauss-Seidel iteration Equ. (4.1) in the block fashion. Notice that the NavP code (Fig. 4.7(b)) is almost identical to the sequential code (Fig. 4.7(a)); the only difference is the *hop*() statement. In the NavP implementation, j and u_1 are agent variables, and other variables are node variables.

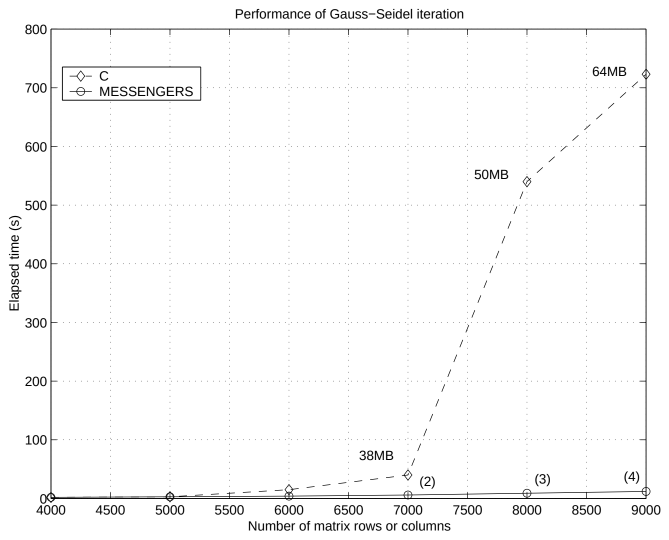


FIG. 4.8. *Performance of Gauss-Seidel iteration.*

Performance data is shown in Fig. 4.8. The K matrices used are banded, and their bandwidth is 10% of their dimensions. The numbers in parentheses by the MESSENGERS curve indicate the number of workstations used, and the amount of total memory required is marked by the C code curve. It would also be possible to implement the NavP version of Gauss-Seidel method using only two workstations, with one pre-fetching the sub-matrix slice

```
(1)  For $j = 1..N$




(2)      For $i = 2..j-1$


(3)          $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$
(4)      End For



(5)      For $i = 1..j-1$
(6)          $T \leftarrow K_{ij}$
(7)          $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(8)          $K_{jj} \leftarrow K_{jj} - TK_{ij}$
(9)      End For
(10) End For
```

(a)

```
(1)    For $j = 1..N$
(1.1)      hop $(node[j])$
(1.2)      If (hopped across node)
(1.3)          inject $(WR(j))$
(1.4)          waitEvent $(IO_{b(j)})$
(1.5)      End If
(1.6)      load (column $j$)
(2)        For $i = 2..j-1$
(2.1)          hop $(node[i])$
(2.2)          load $(K_{ii})$
(3)            $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$
(4)        End For
(4.1)      hop $(node[j])$
(4.2)      unload (column $j$)
(5)        For $i = 1..j-1$
(6)            $T \leftarrow K_{ij}$
(7)            $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(8)            $K_{jj} \leftarrow K_{jj} - TK_{ij}$
(9)        End For
(10)   End For
```

(b)

FIG. 4.9. *Pseudocode for Crout factorization. (a) Sequential. (b) NavP.*

to be used next while the other performing the computation, similar to the implementation of matrix multiplication described in §4.1.

4.3. Crout factorization. When matrix K is symmetric and positive-definite, there exists a non-singular lower triangular matrix L, with unit diagonal entries, and diagonal matrix D such that $K = LDL^T$. The process of obtaining matrices L and D is called *factorization*. One possible way of conducting factorization is due to Crout [11], and the pseudocode is listed in Fig. 4.9(a).

Only the upper part of the matrix K is stored due to its symmetry. To further save storage for a banded matrix, a "skyline" storage scheme is used, in which we link the first non-zero items in every column together to form a "skyline," and we do not store the zero values above this line. Zero values under the skyline are stored, since they could become non-zero during Crout factorization. No additional storage is necessary in the factorization process since the algorithm works in place and K is overwritten by L and D at the end.

In line (3) of the Crout algorithm listed in Fig. 4.9(a), the summation over l corresponds to a dot product of two sub-vectors of columns i and j. These are the two shaded vectors in Fig. 4.10(a). The computation of the j^{th} column depends on the previously computed columns, which constitute the working set. Fig. 4.10(b) is an example for a banded matrix, with the shaded area being the working set for the j^{th} column.

Crout factorization is an algorithm with good locality of access. There does not need to be sufficient memory to hold the entire half matrix; as long as the working set fits in memory, performance is good. However, when the size of the working set exceeds the size of the main memory on a single workstation, extensive thrashing occurs. In Fig. 4.12 the dashed line shows how bad the performance can be when the size of the working set exceeds the size of main memory.

The basic idea of the NavP implementation is for a self-migrating thread to carry the j^{th} column to the working set, which is distributed among several workstations, and compute the terms of the column on these machines. The amount of communication overhead incurred by this approach is obviously much smaller than if the working set is brought in to meet

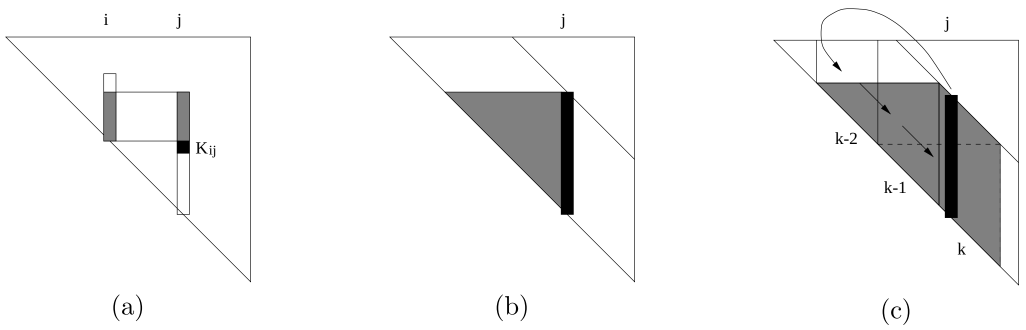


FIG. 4.10. *Crout factorization. (a) Computing of K_{ij} requires the dot product of the two shaded vectors. (b) Working set for column j. (c) Working set decomposition.*

with the j^{th} column. The working set is decomposed into several pieces each consisting of several matrix columns, where the number of pieces is chosen so that each piece can fit into the main memory of one workstation thus avoiding paging overhead. Fig. 4.10(c) shows an example for which the working set is subdivided into three pieces. The arrows indicate how a thread, carrying the j^{th} column which it is computing, moves through the pieces of the working set.

Fig. 4.11(a) shows the network, again assuming that the working set is decomposed into three pieces. The network consists of four nodes. Three of the nodes are used to hold the three working set pieces shown in Fig. 4.10(c), and a fourth node is used to pre-fetch the next piece that will be used later. All four nodes are fully connected, so that a thread can hop to any node from anywhere in one step. As indicated by the arrows, a thread would load from a node variable and carry the j^{th} column in its agent variable, and hop from node 1 to node 3 where piece $k-2$ resides; after it finishes computing using piece $k-2$, it hops forward along the link to node 2 where piece $k-1$ resides (note that $k-2$ and $k-1$ are pieces that have been already computed in the previous loops); finally, it hops back to node 1, computes the rest of column j using piece k, and unloads the j^{th} column from its agent variable to the node variable, and then scales the j^{th} column using the diagonal terms it carries back from the other nodes. These loops go on and on until all columns in piece k are computed, at which time the thread moves on to the next node (4 in the example) to compute piece $k+1$. Again, the previously computed pieces $k-1$ and k will be used in computing $k+1$, but piece $k-2$ will no longer be used in factorization, so we can now save piece $k-2$ to the hard disk, and use node 3 to pre-fetch piece $k+2$. Fig. 4.11(b) shows this next step when piece $k+1$ is being computed. If we compare Fig. 4.11(b) with Fig. 4.11(a), we see that the network is like a "running wheel" rotating forward (clockwise) while it processes the matrix pieces sequentially. In order for a thread to hop to the right machine for a column, a map of a column number to a node number is kept on every node (*node*[.] in Fig. 4.9(b)). This map is a by-product of data distribution.

Two threads are employed in our implementation. The first one, named *Factor*, has its code (listed in Fig. 4.9(b)) augmented from the original Crout algorithm (Fig. 4.9(a)). Three *hop*()s (lines (1.1), (2.1), and (4.1)) and three *load*()/*unload*()'s (lines (1.6), (2.2), and (4.2)) are added. A *load*() statement copies data from a node variable to an agent variable, and a *unload*() statement does the opposite operation. The cost of the *hop*() statement is negligible if the destination node happens to be the one that the thread resides on. A second thread called *WR* (writer and reader) is injected (line (1.3)) to conduct post-writing and pre-fetching whenever *Factor* hops across node boundary (lines (1.1) and (1.2)). *WR*(j) would hop back to the node from which *Factor* came, write the factorized piece to disk,

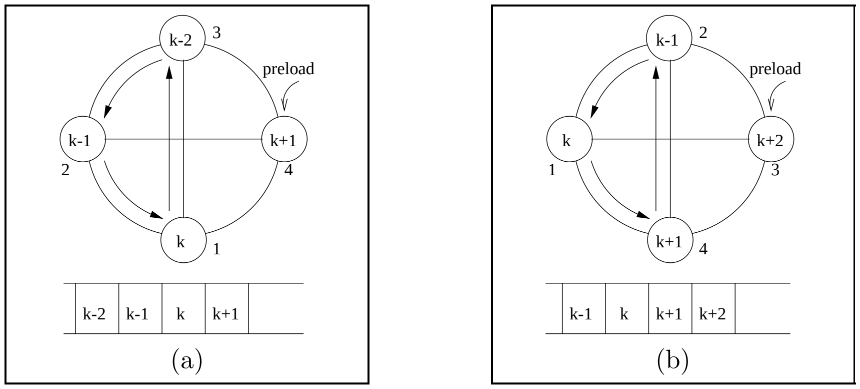


FIG. 4.11. *The network for Crout factorization. (a) currently computing piece k. (b) currently computing piece $k+1$.*

read in the piece $b(j)+1$, which is next to the one containing column j, and then signal an event $IO_{b(j)+1}$ that indicates that it has done the appropriate fetching. This event is caught by *Factor* at line (1.4), when it visits the node next time, to ensure correctness of the computation. The pseudocode for *WR* is similar to the one in Fig.4.1(c) and therefore not listed. We gain a huge advantage doing fetching not only from the pipelining of disk I/O, but also from the fact that now our program can solve problems that are many times (35 in our example) larger than the memory available on a single workstation with only few workstations (4 in our example). We would have had to use 35 machines had we not used fetching.

Fig. 4.12 shows the performance of Crout factorization. The K matrices used are banded, and their bandwidth is 10% of their dimensions. The numbers in parentheses by the MESSENGERS curve indicate the number of pieces into which the matrix K is subdivided. Total memory required is also marked for some problem sizes. One may notice that the gap between the two curves (excluding the segment corresponding to disk thrashing) now is larger than what's shown in Fig. 4.4 (matrix multiplication case). This is because the amount of communication through the network is larger. In fact, since almost all the columns (except for the columns in the very first piece) are carried around, the amount of communication is proportional to the size of the entire matrix ($\Theta(N^2)$). However, since the computational complexity of Crout factorization is $\Theta(N^3)$, our implementation is scalable. Also, with fast network (e.g., 100Mbps), our analysis and test show that the time for communication overhead can be reduced to about 5% of the total elapsed time.

Fig. 4.13 shows the pseudocode of Crout factorization using MP. Notice that in this pseudocode fetching is not considered at all, and some details, e.g., the boundary cases of k, are left out. Also, this code is written assuming that the matrix K is decomposed into P pieces with each working set decomposed into three pieces. More **Else If** constructs would be needed if working set is sliced into more pieces. In the pseudocode, μ is the process ID, which is defined as the index of a matrix piece that a processor owns. I_k is the index of the first column that piece k owns. And $\{K_{dd}\}$ is a vector of diagonal entries. If we compare the MP implementation shown in Fig. 4.13, with the original algorithm shown in Fig. 4.9(a), the differences are considerable. The **If**/**Else If** constructs artificially break the original code into blocks that will be executed on different nodes. In Fig. 4.13, code lines (6)–(8), (6')–(8'), and (6")–(8") used to be one loop in the original algorithm (lines (2)–(4) in Fig. 4.9(a)), or the NavP code (lines (2)–(4) in Fig. 4.9(b)), but they are broken up by MP into different sub-blocks. In contrast, NavP-based DSC implementation (Fig. 4.9(b)) keeps the original code structure unchanged.

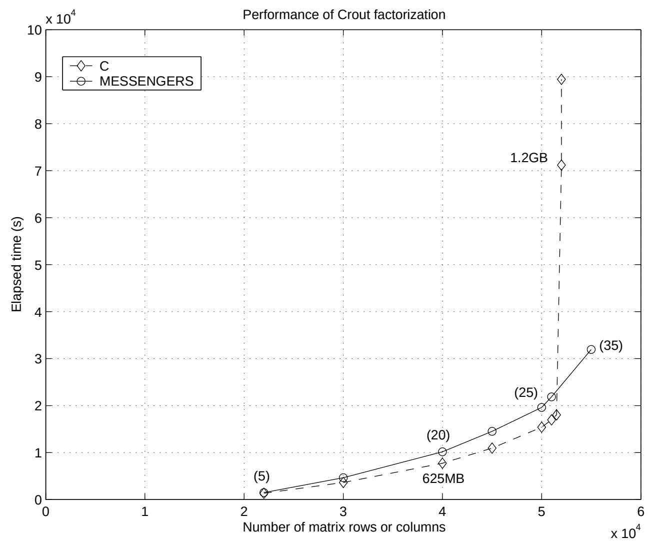


FIG. 4.12. *Performance of Crout factorization.*

```
(1)  For $k = 1 .. P$
(2)    If $\mu == k$
(3)      For $j = I_k .. I_{k+1} - 1$
(4)        Send $(col\, j,\ k-2)$
(5)        Recv $(col\, j,\ \{K_{dd}\},\ k-1)$
(6)        For $i = I_k .. j - 1$
(7)          $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li} K_{lj}$
(8)        End For
(9)        For $i = 1 .. j - 1$
(10)         $T \leftarrow K_{ij}$
(11)         $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(12)         $K_{jj} \leftarrow K_{jj} - T K_{ij}$
(13)       End For
(14)     End For
(15)   Else If $\mu == k - 2$
(16)     For $m = I_k .. I_{k+1} - 1$
(17)       Recv $(col\, j,\ k)$
(6')       For $i = I_{k-2} .. I_{k-1} - 1$
(7')         $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li} K_{lj}$
(8')       End For
(18)       Send $(col\, j,\ \{K_{dd}\},\ k-1)$
(19)     End For
(20)   Else If $\mu == k - 1$
(21)     For $m = I_k .. I_{k+1} - 1$
(22)       Recv $(col\, j,\ \{K_{dd}\},\ k-2)$
(6")       For $i = I_{k-1} .. I_k - 1$
(7")         $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li} K_{lj}$
(8")       End For
(23)       Send $(col\, j,\ \{K_{dd}\},\ k)$
(24)     End For
(25)   End If
(26) End For
```

FIG. 4.13. *Pseudocode for Crout factorization in MP (without fetching).*

5. Other Approaches. Using a network of workstations as a data paging farm is a promising notion because future improvements in the performance of disk paging appear to be limited by the inherent bottleneck of mechanical seek time, but there is no such limit on the speed and bandwidth of computer network. The "remote memory paging" approach [3, 4], as a special case of DSM, is based on the notion and is one way of doing DSC. In this approach, a stationary process runs on a single machine and accesses data remotely through the network. A major disadvantage of this approach is its non-scalability because the principle of pivot-computes is clearly violated. A positive feature of this approach is that it is a new paging scheme at operating system level which means it does not assume any knowledge of the specifics of any applications.

The approach of DSC using self-migrating threads distinguishes itself from remote memory paging in three ways. First, we do not move all data to one single machine; rather we move computation to large-sized data, which can significantly reduce communication overhead. Second, we use auxiliary threads (*WR* in two of our examples) with very simple behavior and synchronization to utilize the "spare" CPU cycles in the workstation farm to pipeline data I/O. This enables us to use only a few workstations to solve very large problems, with total data size much larger than the total amount of main memory available in the workstation farm. Third, we augment a sequential program to make it self-migrate in the network; this requires knowledge of data accessing pattern of the application.

The MP approach to DSC is scalable, but it achieves computation mobility by mimicing strong mobility manually. In contrast, the NavP approach passes the burden of code restructuring to a mobile agent compiler that supports strong mobility. Therefore, our approach is much easier than MP.

Another possible approach to DSC is the "remote procedure call," or RPC. Indeed, RPC can suggest pivot-computes too. Of course, pivot-computes requires that services are data-distribution driven, rather than being provided transparently by a middleware based on availability. But RPC and NavP approach use different metaphors. In fact, RPC is a special case of our approach. A remote procedure call will always have to return the control to the calling client, while a self-migrating thread does not have to return to the location where it is spawned. This could cause difficulties to RPC in handling certain applications (e.g., ones that have locus of computation move through a linear data structure like in Crout factorization). Although some RPC-based systems [9] have figured out ways to pass intermediate data from server to server directly without going through the client unnecessarily, method invocations will still have to be through the calling client. Rather than resolving these issues stemming from a metaphor that is improper for some applications, it is easier to directly embrace the NavP approach in general purpose distributed sequential programming.

6. Final Remarks. We have presented three case studies of distributed sequential computing on a network of workstations. In each case, the performance on a large problem, with a total size of data considerably larger than the memory of a single machine, is quite close to the performance that would be achieved if the data fit in memory. Scalability is achieved by following the principle of pivot-computes using computation mobility facilitated by self-migrating threads. In addition, through the use of a mobile agent compiler that supports strong mobility, our implementations preserve algorithmic integrity. The algorithms remain essentially unchanged: the main difference between our DSC implementation and the sequential algorithm is the addition of *hop*() statements to allow the computations to migrate in the distributed environment. Algorithmic integrity not only makes program development and maintenance easy, but also enables the DSC programs to be backward compatible with uni-processor architecture. Since the sequential code structure is unchanged in the NavP implementation, if the navigational statements are ignored, the NavP program runs correctly on a uni-processor machine. This is because data location information is referenced in the *hop*() statements (i.e., navigational statements), but not in any other statements. Backward compatibility to a uni-processor is more problematic in the case of MP code, since data location information is explicitly used in changing the code structure.

One may have noticed that in all our examples we use, as a starting point for the DSC implementation, a sequential algorithm with good locality of access. This makes it easy to insert *hop*() statements so that the resulting NavP program has relatively coarse granularity. While it might appear that this places special constraints on the NavP approach, we argue that this is not the case, for two different reasons. First, in most real-world situations, the sequential program will already have been written to take advantage of locality of access, so it will not be necessary to modify the sequential code before adding the navigational statements. Second, any fine-grained algorithm needs to be "coarsened" before it can be efficiently implemented in a distributed memory environment using any approach such as

MP. For example, in matrix algorithms, operations on scalars generally need to be replaced by operations on blocks in the matrix.

For programs that use large data sets, paging may become an impediment to scalability. NavP-based DSC provides a means of eliminating this paging overhead with only a minor amount of programming effort. Of course, further improvement can be usually gained from a parallel re-implementation. In some cases the parallel implementation is an easy modification of the sequential algorithm (e.g., the iterative method), while in other cases a parallel implementation may require significant rethinking and reworking of the sequential algorithm (e.g., Crout factorization). In any specific situation, the question of whether the additional improvement to be gained from a parallel re-implementation justifies what might be a major re-programming effort needs to be evaluated on a case-by-case basis. NavP-based DSC code provides a good starting point for incremental parallelization. Some preliminary work on using NavP-based DSC to facilitate distributed parallel programming is presented in [20, 21].

DSC frequently needs to perform fetching of data. This is seen in two of our examples. In DSC matrix multiplication, there are p^2 distinct sub-matrix pairs that must be multiplied. In DSC Crout factorization, total matrix size (e.g., 1.2GB) is much larger than the size of the working set (e.g., 64MB). In both cases, many more workstations would have been used had we not employed fetching. Fetching is quite easy in the NavP approach. Sequential NavP programs keep their original code structures, the threads doing fetching have very simple tasks, their code is separated from the code for real computations, and their synchronization with the main computation locus is straightforward. In contrast, with MP the code lines for computation and fetching are unavoidably tangled.

As described in this chapter, DSC using NavP requires inserting explicit navigational statements in the programs for computation to follow data. It would be useful if the navigational statements could be derived directly from data distribution and hence made implicit. This problem is related to recent research on thread migration in DSM [24, 12]. Some preliminary approaches to navigational programming in DSM are presented in [22].

REFERENCES

[1] J.-Y. Berthou and L. Colombet, *Which approach to parallelizing scientific codes—that is the question*, Parallel Computing, 23 (1997), pp. 165–179.

[2] W. L. Briggs, *A Multigrid Tutorial*, Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1987.

[3] G. Dramimitnos and E. P. Markatos, *Adaptive and reliable paging to remote main memory*, Journal of Parallel and Distributed Computing, 58 (1999), pp. 357–388.

[4] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets, *Cashmere-VLM: Remote memory paging for software distributed shared memory*, in Proceedings, 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, Los Alamitos, Calif., Apr. 1999, IEEE Computer Society, pp. 153–159.

[5] A. Fuggetta, G. P. Picco, and G. Vigna, *Understanding code mobility*, IEEE Transactions on Software Engineering, 24 (1998), pp. 342–361.

[6] M. Fukuda, L. F. Bic, and M. B. Dillencourt, *Messages versus messengers in distributed programming*, Journal of Parallel and Distributed Computing, 57 (1999), pp. 188–211.

[7] E. Gendelman, Messengers *User's Manual (version 2.1)*, Information & Computer Science, University of California, Irvine, 2001.

[8] E. Gendelman, L. F. Bic, and M. B. Dillencourt, *Fast file access for fast agents*, in Proceedings, 5th International Conference on Mobile Agents, MA 2001, G. P. Picco, ed., vol. 2240 of Lecture Notes in Computer Science, Berlin, Germany, Dec. 2001, Springer-Verlag, pp. 88–102.

[9] A. S. Grimshaw, *Object-oriented parallel processing with Mentat*, Information Sciences, 93 (1996), pp. 9–34.

[10] W. D. Gropp, *Learning from the success of MPI*, in Proceedings, 8th International Conference on High Performance Computing - HiPC 2001, B. Monien, V. K. Prasanna, and S. Vajapeyam, eds., vol. 2228 of Lecture Notes in Computer Science, Berlin, Germany, Dec. 2001, Springer-Verlag, pp. 81–92.

[11] T. J. R. Hughes, *The Finite Element Method : Linear Static and Dynamic Finite Element Analysis*, Prentice Hall, Englewood Cliffs, N.J., 1987.

[12] H. Jiang and V. Chaudary, *MigThread: Thread migration in DSM systems*, in Proceedings, International Conference on Parallel Processing Workshop on Compile/Runtime Techniques for Parallel Computing, Alamitos, Calif., Aug. 2002, IEEE Computer Society, pp. 581–588.

[13] D. Kotz, R. Gray, and D. Rus, *Future directions for mobile agent research*, IEEE Distributed Systems Online, 3 (2002).

[14] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin Cummings, Redwood City, Calif., 1994.

[15] D. B. Lange and M. Oshima, *Seven good reasons for mobile agents*, Communications of the ACM, 42 (1999), pp. 88–89.

[16] C. Leopold, *Parallel and Distributed Computing: A Survey of Models, Paradigms, and Approaches*, John Wiley & Sons, New York, 2001.

[17] D. Milojicic, *Trend wars: Mobile agent applications*, IEEE Concurency, 7(3) (1999), pp. 80–90.

[18] R. Morgan, *Building an Optimizing Compiler*, Butterworth-Heinemann, Boston, Mass., 1998.

[19] L. Pan, L. F. Bic, and M. B. Dillencourt, *Shared variable programming beyond shared memory: Bridging distributed memory with mobile agents*, in Proceedings of the 6th International Conference on Integrated Design & Process Technology (IDPT-2002), H. Ehrig, B. Kramer, and A. Ertas, eds., Grandview, Texas, June 2002, Society for Design & Process Science.

[20] L. Pan, L. F. Bic, M. B. Dillencourt, J. J. Huseynov, and M. K. Lai, *Distributed parallel computing using navigational programming: Orchestrating computations around data*, in Proceedings, 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002), Calgary, AB, Canada, Nov. 2002, ACTA Press, pp. 458–463.

[21] L. Pan, L. F. Bic, M. B. Dillencourt, and M. K. Lai, *From distributed sequential computing to distributed parallel computing*, in Proceedings of the 2003 ICPP Workshop on High Performance Scientific and Engineering Computing with Applications (HPSECA-03), Los Alamitos, Calif., Oct. 2003, IEEE Computer Society.

[22] L. Pan, M. K. Lai, J. J. Huseynov, L. F. Bic, and M. B. Dillencourt, *Facilitating agent navigation using DSM – high level designs*, in Proceedings of the 7th International Conference on Integrated Design & Process Technology (IDPT-2003), A. Ertas, ed., Grandview, Texas, June 2003, Society for Design & Process Science.

[23] A. S. Tanenbaum and M. Van Steen, *Distributed Systems Principles and Paradigms*, Prentice Hall, Upper Saddle River, N.J., 2002.

[24] G. Vallée, C. Morin, J.-Y. Berthou, I. D. Malen, and R. Lottiaux, *Process migration based on Gobelins distributed shared memory*, in Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid CCGRID 2002, Alamitos, Calif., May 2002, IEEE Computer Society, pp. 301–306.

[25] C. Wicke, L. F. Bic, M. B. Dillencourt, and M. Fukuda, *Automatic state capture of self-migrating computations in* MESSENGERS, in Proceedings, Second International Conference on Mobile Agents, MA '98, K. Rothermel and F. Hohl, eds., vol. 1477 of Lecture Notes in Computer Science, Berlin, Germany, Sept. 1998, Springer-Verlag, pp. 68–79.